Universidade Federal Fluminense
Instituto de Ciências Exatas - ICEx

Computação de Alto Desempenho 2

Paralelização da solução para equação de Schrödinger

Bianca Maia

23 de julho de 2022

Resumo

Ao reconhecer a necessidade de aumentar o poder computacional para o interesse acadêmico em pesquisas científicas, descobertas e também otimizações na operação de uma grande quantidade de dados, um novo tipo de máquinas foi desenvolvido, estas são chamadas de supercomputadores. A computação de alto desempenho (HPC) é o uso ativo desses supercomputadores e clusters para a operação de tarefas que exigem grande poder de processamento e problemas em que a solução numérica é muito complicada de se obter. A maioria desses problemas numéricos contém seus dados dentro de matrizes, portanto, uma forma de melhorá-los com computação paralela é por unidades de processamento gráfico (GPU) que possuem uma arquitetura paralela massiva que o torna extremamente eficaz nas operações com matrizes.

Um exemplo bem conhecido de um problema complicado, em relação ao tempo de processamento, são as equações diferenciais parciais. Este trabalho tem como foco resolver uma equação de Schrödinger pelo método das diferenças finitas, otimizando seu tempo de execução com CUDA. Apresenta também técnicas de melhoria de código e análise de flags.

Palavras-chaves: HPC, computação paralela, programação paralela com GPU, CUDA

Abstract

By recognizing the urge to increase computational power for academic interest in scientific research, discoveries and also optimizations in operation some large amount of data, a new type of machines had been developed, these are called supercomputers. High-perfomance computing (HPC) is the active use of these supercomputers and cluster in order to operating tasks that require a large processing power and problems in which the numerical solution are too complicated to obtain. Most of these numerical problems contains its data inside matrices, therefore a way of improving it with parallel computing it's by graphics processing units (GPU) which have a massive parallel architecture that makes it extremely effective in matrices operations.

A well-known example of a complicated problem, in respect of processing time, is the partial differential equations. This works focus on solving a Schrödinger equation by the finite difference method, optimizing its execution time with CUDA. It also presents techniques of code improvement and flags analysis.

Keywords: HPC, parallel computing, GPU parallel programming, CUDA

Conteúdo

1 Introdução

Considere a equação de Schrödinger depedente do tempo em duas dimensões que descreve como o estado quântico de um sistema física evolui com o tempo

$$-\frac{\hbar}{2m}\nabla^2\psi(\vec{r}) + V(\vec{r})\psi(\vec{r}) = E\psi(\vec{r}) \tag{1}$$

onde $\psi(\vec{r}) = \psi(x, y, t)$, $V(\vec{r}) = V(x, y)$ e $\nabla^2 = \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2}$.

O objetivo do algoritmo é resolver um problema onde um eletron está confinado dentro de um potencial quadrático com a sua posição inicial dada pela gaussiana:

$$\psi(x, y, t = 0) = e^{ik_{0x}x}e^{ik_{0y}y}exp\left(-\frac{(x - x_0)^2}{2\sigma_0^2}\right)exp\left(-\frac{(y - y_0)^2}{2\sigma_0^2}\right) \tag{2}$$

O potencial utilizado para a resolução computacional desse problema foi:

$$V(x, y) = 0.01x^2 + 0.07y^2$$

O programa para a solução dessa equação com essas condições foi inicialmente desenvolvido na linguagem C em serial, dispensando quaisquer preocupações em relação ao seu tempo de execução. Posteriormente, esse foi otimizado por programação paralela utilizando o openMP com o objetivo de reduzir o tempo de compilação.

2 Objetivo

Resolver uma equação diferencial parcial computacionalmente pode ser custoso em termos de tempo de compilação dependendo das condições iniciais determinadas para o problema.

O principal objetivo do projeto é reduzir o tempo de execução do programa com o uso da programação paralela, sem perder as caracteristícas do problema da equação de Schrödinger.

Na finalidade de atingir essa otimização, serão analisados os desempenhos do programa serial para um conjunto de flags referentes à dois compiladores em dois hardwares diferentes. As melhores flags serão aproveitadas para uma análise do programa em paralelo para um conjunto de threads (referentes à cada máquina).

Na necessidade de otimizações à nível código no serial, um profiling apontará quais dependências do programa serial mais consomem desempenho para aplicar as boas práticas de programação eficientemente.

Finalmente, com as melhores flags e as melhores threads, o programa paralelizado com openMP poderá obter uma otimização de tempo mais agressiva.

3 Metodologia

A metodologia usada para resolver o problema de equação diferencial parcial foi o **método das diferenças finitas** [1].

3.1 Discretização para a equação de Schrödinger

Reescrevendo a equação (1) como

$$\psi(x,y,t) = U(t)\psi(x,y,t=0) \tag{3}$$

onde $U(t) = e^{-i\tilde{H}t}$ e $\tilde{H} = -\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} + V(x,y)$. Temos que $U(t)$ é um operador que desenvolve a função de onda por um determinado tempo t e $\tilde{H}$ é o operador Hamiltoniano.

Assumindo que o pacote de onda avançará de acordo com um intervalo temporal definido por Δt

$$\psi_{i,j}^{n+1} = U(\Delta t)\psi_{i,j}^{n} \tag{4}$$

o superescrito denota o tempo $t = n\Delta t$ e os sobrescritos denotam variáveis espaciais $x = i\Delta x$ e $y = i\Delta y$.

Dada a inversa que retorna à solução no intervalo anterior

$$\psi^{n-1} = U^{-1}(\Delta t)\psi^{n} \tag{5}$$

e a relação de diferença entre ψ^{n+1} e ψ^{n-1}

$$\psi^{n+1} = \psi^{n-1} + [e^{-i\tilde{H}\Delta t} - e^{+i\tilde{H}\Delta t}]\psi^{n} \tag{6}$$

da expansão de Taylor,

$$\frac{\partial^2\psi}{\partial x^2} \approx \frac{-1}{2}[\psi_{i+1,j}^{n} + \psi_{i-1,j}^{n} - 2\psi_{i,j}^{n}] \tag{7}$$

logo,

$$\psi_{i,j}^{n+1} = \psi_{i,j}^{n-1} - 2i[(4\alpha + \frac{1}{2}\Delta t V_{i,j})\psi_{i,j}^{n} - \alpha(\psi_{i+1,j}^{n} + \psi_{i-1,j}^{n} + \psi_{i,j+1}^{n} + \psi_{i,j-1}^{n})] \tag{8}$$

onde $\alpha = \Delta t/2(\Delta x)^2$.

Para providenciar a conservação numérica da função densidade de probabilidade, o método da diferenças finitas resolve a parte real e a parte imaginária [2]. Portanto,

$$R_{i,j}^{n} = R_{i,j}^{n-1} + 2[(4\alpha + \frac{1}{2}\Delta t V_{i,j})I_{i,j}^{n} - \alpha(I_{i+1,j}^{n} + I_{i-1,j}^{n} + I_{i,j+1}^{n} + I_{i,j-1}^{n})]$$

$$I_{i,j}^{n} = I_{i,j}^{n-1} - 2[(4\alpha + \frac{1}{2}\Delta t V_{i,j})R_{i,j}^{n} + \alpha(R_{i+1,j}^{n} + R_{i-1,j}^{n} + R_{i,j+1}^{n} + R_{i,j-1}^{n})]$$

4 Implementação (fluxograma)

O fluxograma do código utilizado para resolver o problema da equação de Schrödinger está disposto abaixo.

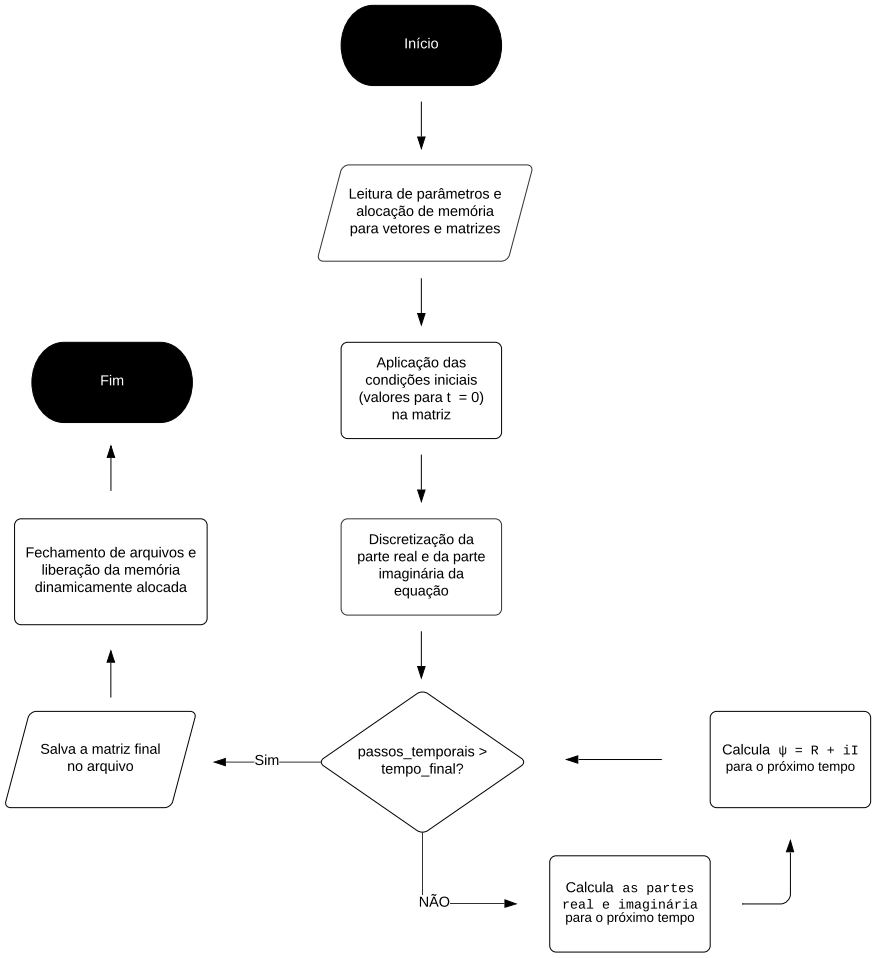

Figura 1: Fluxograma do programa para resolver a equação de Schrödinger com o método das diferenças finitas.

5 Benchmark (serial)

O benchmark foi realizado apenas para o Lab 107 e o SequanaX, devido à indisponibilidade do B710 durante o período em que essas análises foram realizadas.

5.1 Lab 107

Nesse computador as flags utilizadas para o compilador gnu foram:

1. -OX
2. -OX -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math
3. -OX -mtune=native -march=native -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math
4. -OX -mtune=corei7-avx -march=corei7-avx -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math

e para o compilador ifort:

1. -OX
2. -OX -w -mp1 -zero -xHOST
3. -OX -w -mp1 -zero -xHOST -fast=2

onde as otimizações -OX compreendem a variação entre -O0, -O1, -O2 e -O3.

5.1.1 Análise para o compilador gnu

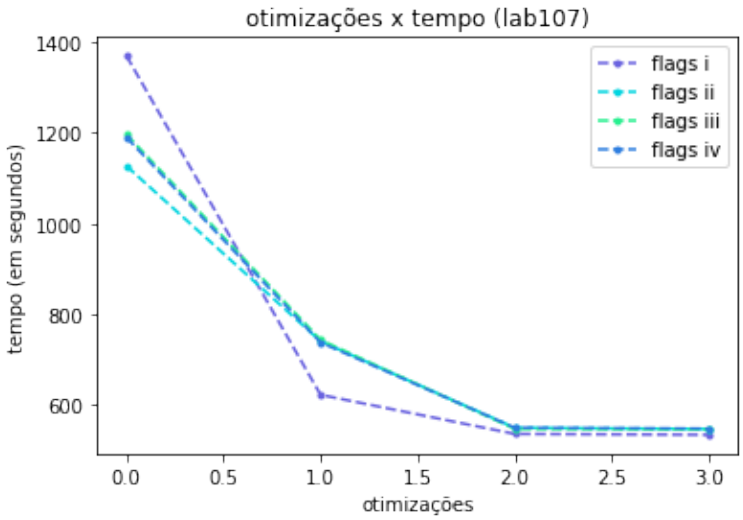

Figura 2: Gráfico para os tempos em função das otimizações com as flags do compilador gnu no computador do Lab 107.

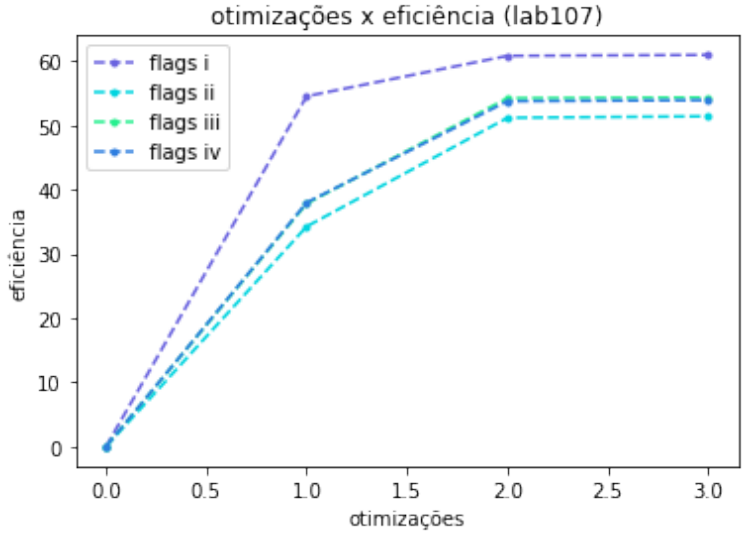

Figura 3: A eficiência utilizando as flags do compilador gnu no Lab 107.

É facilmente observável que a melhor flag gnu para esse computador

é a -O2 do conjunto de flags i. Embora a -O3 do mesmo conjunto seja uma concorrente em eficiência, pelo gráfico do tempo vemos que a outra supera essa última.

5.1.2 Análise para o compilador ifort

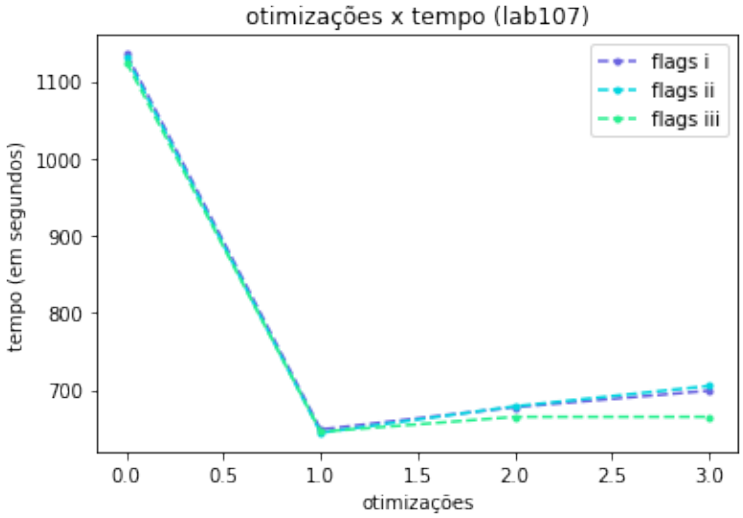

Figura 4: O tempo para o compilador ifort no Lab 107.

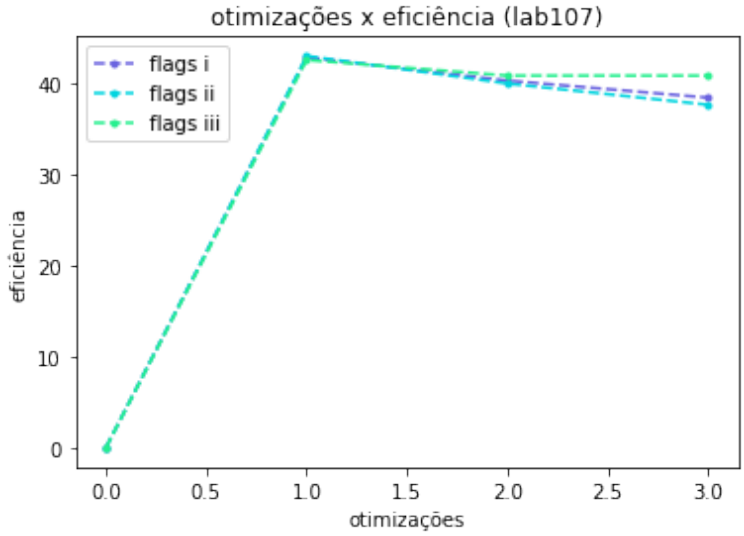

Figura 5: Gráfico da eficiência para o compilador ifort no Lab 107.

Nesse compilador, é imediato que a melhor flag é a -O1 para quaisquer um dos conjuntos de flags utilizados.

5.2 SequanaX

Nesse computador as flags utilizadas para o compilador gnu foram:

1. -OX
2. -OX -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math
3. -OX -mtune=native -march=native -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math
4. -OX -mtune=core-avx2 -march=core-avx2 -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math

e para o compilador ifort:

1. -OX
2. -OX -w -mp1 -zero -xHOST
3. -OX -w -mp1 -zero -xHOST -fast=2

onde as otimizações -OX compreendem a variação entre -O0, -O1, -O2 e -O3.

5.2.1 Análise para o compilador gnu

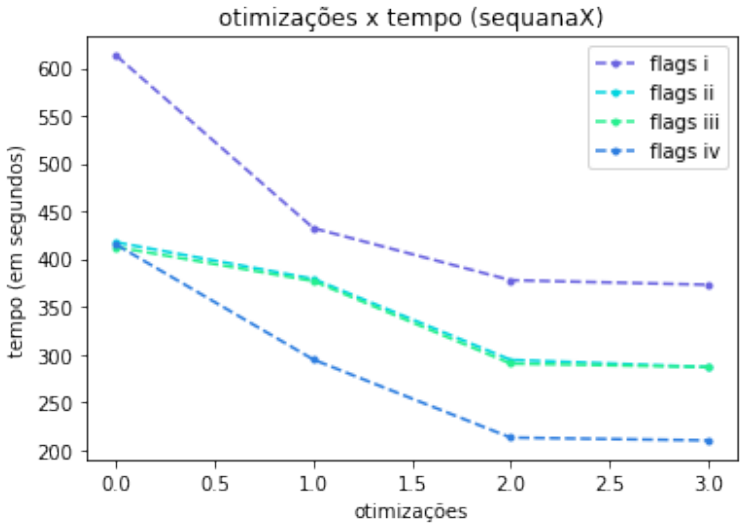

Figura 6: O tempo com as flags gnu para o SequanaX.

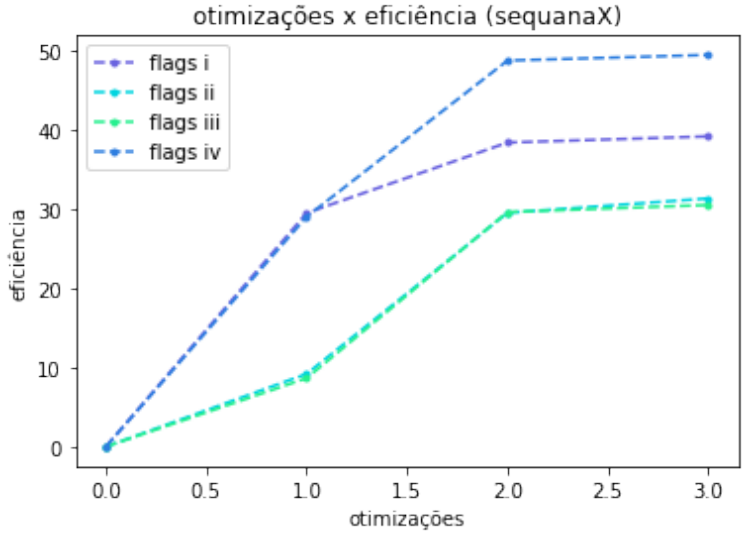

Figura 7: Gráfico da eficiência para o compilador gnu no SequanaX.

Diferentemente da situação anterior para o compilador gnu, as melhores flags são aquelas acompanhadas do último conjunto de flags que é acompanhando

de flags referentes à arquitetura do computador. Nesse conjunto, tanto a -O2 quanto a -O3 são as mais eficientes que todas as outras, embora a -O3 ligeiramente apresente um desempenho superior à -O2.

5.2.2 Análise para o compilador ifort

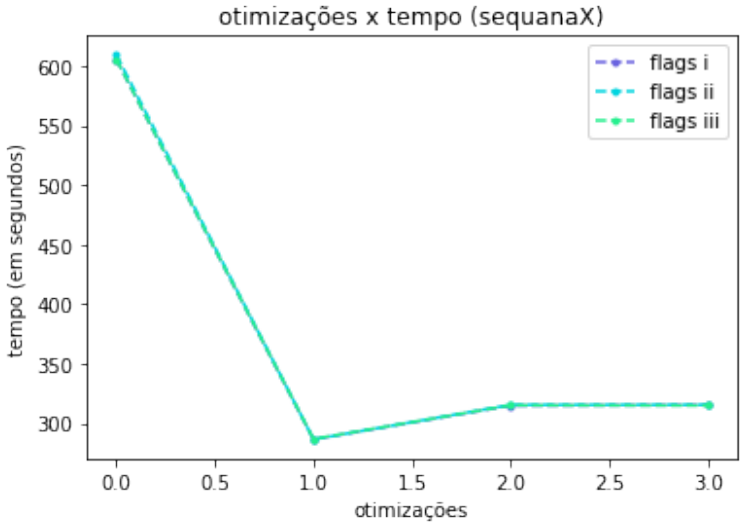

Figura 8: Tempo para o compilador ifort no SequanaX.

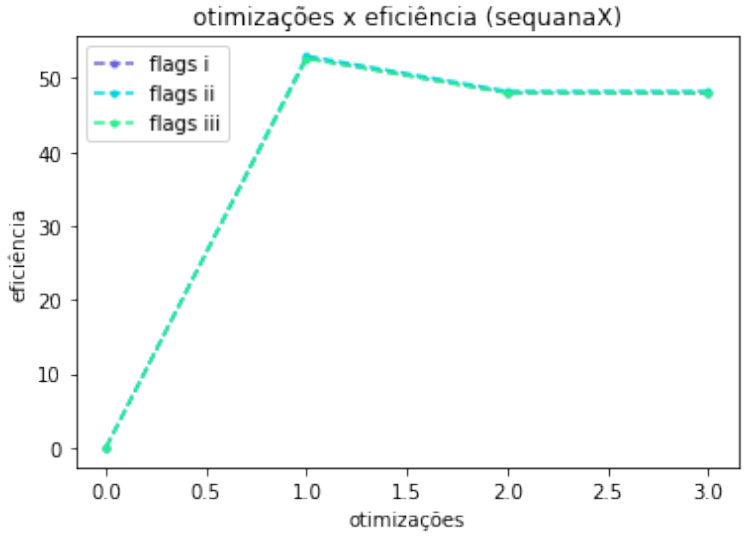

Figura 9: A eficiência para as flags ifort no SequanaX.

Nesse último caso, as curvas são praticamente concorrentes. As melhores flags são aquelas com otimização -O1 para todos os conjuntos utilizados nesse computadores.

5.3 Comentários

Para o compilador gnu, a melhor flag na máquina do Lab 107 é a:

```
-lm -O2
```

Por outro lado, no SequanaX o impacto das otimizações de arquitetura são mais intensos e portanto a melhor flag é:

```
-lm -O3 -mtune=core-avx2 -march=core-avx2
-fexpensive-optimizations -m64 -foptimize-register-move
-funroll-loops -ffast-math
```

Para o compilador ifort, em ambos os computadores as melhores flags são aquelas acompanhadas da otimização -O1 independente do conjunto de flags. Não há critérios rigorosos para decidir qual conjunto utilizar na análise do código em paralelo, dado que a diferença de tempo entre esses é ignorável.

6 Profile (análise dos gargalhos)

6.1 Profile sem flags

O profile foi executado para o programa serial do projeto com a matriz reduzida. O tempo de execução do programa está logo abaixo, antes dos resultados obtidos pelo profiling.

```
real 35m15,495s
user 35m5,720s
sys  0m4,657s
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.30   2117.86  2117.86   600000     3.53     3.53
    finite_difference
  0.00   2117.90     0.04                             main
  0.00   2117.92     0.02        1    20.07    20.07  wave_packet
  0.00   2117.92     0.00        3     0.00     0.00  allocate
  0.00   2117.92     0.00        1     0.00     0.00
    allocate_complex
  0.00   2117.92     0.00        1     0.00     0.00  potential

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
     else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
     function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
     the function in the gprof listing. If the index is
     in parenthesis it shows where it would appear in
     the gprof listing if it were to be printed.


Copyright (C) 2012-2020 Free Software Foundation, Inc.

```

```
Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.


         Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.00% of 2117.92 seconds

index % time    self  children    called     name
                                             <spontaneous>
[1]     100.0    0.04 2117.88                 main [1]
             2117.86    0.00  600000/600000      finite_difference [2]
                0.02    0.00       1/1           wave_packet [3]
                0.00    0.00       3/3           allocate [4]
                0.00    0.00       1/1           allocate_complex [5]
                0.00    0.00       1/1           potential [6]
-----------------------------------------------
             2117.86    0.00  600000/600000      main [1]
[2]     100.0 2117.86    0.00  600000         finite_difference [2]
-----------------------------------------------
                0.02    0.00       1/1           main [1]
[3]       0.0    0.02    0.00       1         wave_packet [3]
-----------------------------------------------
                0.00    0.00       3/3           main [1]
[4]       0.0    0.00    0.00       3         allocate [4]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[5]       0.0    0.00    0.00       1         allocate_complex [5]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[6]       0.0    0.00    0.00       1         potential [6]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
     index  A unique number given to each element of the table.
    Index numbers are sorted numerically.
    The index number is printed next to every function name so
    it is easier to look up where the function is in the table.

     % time This is the percentage of the `total' time that was spent
    in this function and its children.  Note that due to
```

```
    different viewpoints, functions excluded by options, etc,
    these numbers will NOT add up to 100%.

     self This is the total amount of time spent in this function.

     children This is the total amount of time propagated into this
    function by its children.

     called This is the number of times the function was called.
    If the function called itself recursively, the number
    only includes non-recursive calls, and is followed by
    a `+' and the number of recursive calls.

     name The name of the current function.  The index number is
    printed after it.  If the function is a member of a
    cycle, the cycle number is printed between the
    function's name and the index number.


For the function's parents, the fields have the following meanings
    :

     self This is the amount of time that was propagated directly
    from the function into this parent.

     children This is the amount of time that was propagated from
    the function's children into this parent.

     called This is the number of times this parent called the
    function `/' the total number of times the function
    was called.  Recursive calls to the function are not
    included in the number after the `/'.

     name This is the name of the parent.  The parent's index
    number is printed after it.  If the parent is a
    member of a cycle, the cycle number is printed between
    the name and the index number.

If the parents of the function cannot be determined, the word
`<spontaneous>' is printed in the `name' field, and all the other
fields are blank.

For the function's children, the fields have the following
    meanings:

     self This is the amount of time that was propagated directly
    from the child into the function.

     children This is the amount of time that was propagated from
    the
    child's children to the function.

     called This is the number of times the function called
    this child `/' the total number of times the child
    was called.  Recursive calls by the child are not
    listed in the number after the `/'.

```

```
     name This is the name of the child.  The child's index
    number is printed after it.  If the child is a
    member of a cycle, the cycle number is printed
    between the name and the index number.

 If there are any cycles (circles) in the call graph, there is an
 entry for the cycle-as-a-whole.  This entry shows who called the
 cycle (as parents) and the members of the cycle (as children.)
 The `+' recursive calls entry shows the number of function calls that
 were internal to the cycle, and the calls entry for each member shows,
 for that member, how many times it was called from other members of
 the cycle.


Copyright (C) 2012-2020 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.


Index by function name

   [4] allocate                [2] finite_difference       [6] potential
   [5] allocate_complex        [1] main                    [3] wave_packet
```

6.2 Profile com a melhor flag

A matriz é a mesma do profile anterior.

```
real  10m32,015s
user  10m30,145s
sys 0m1,260s
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.40    633.97   633.97                             finite_difference
  0.00    633.98     0.01                             wave_packet

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
```

```
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
     else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
     function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
     the function in the gprof listing. If the index is
     in parenthesis it shows where it would appear in
     the gprof listing if it were to be printed.


Copyright (C) 2012-2020 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification
    ,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
```

6.3 Análise de resultados

A função que mais consome processamento é a função responsável por resolver a equação utilizando o método das diferenças finitas.

O profile sem flags não resolve o problema de tempo de execução, é mais eficiente com as flags que provocam uma melhoria de aproximadamente 20min.

7 Técnicas de otimização de software

Como demonstrado no profile, a função responsável pelo consumo de tempo é a função do método das diferenças finitas. Nada é possível fazer a respeito da otimização do código dessa função, no entanto ainda é viável conferir outras funções essenciais para a resolução do problema, a função do potencial e a do pacote de onda.

```
1  void potential(double **potential){
2      int i, j;
3      double n = 110;
4
5      for(i = 0; i < size; i++){
6          for(j = 0; j < size; j++)
7              potential[i][j] = 0.07*pow(i, 2) + 0.01*pow(j, 2);
8
9      }
10 }
11
12 void wave_packet(double complex **p, double **im, double **r,
       double k0x, double k0y, double x0, double y0, double sigma){
13     int x, y, it = 0;
14
15     for(x = 0; x < size; x++){
16         for(y = 0; y < size; y++){
17             r[x][y] = cos(k0x*x + k0y*y)*exp(-(pow(x - x0, 2) + pow
       (y - y0, 2))/(2*pow(sigma, 2)));            im[x][y] = sin(k0x*
       x + k0y*y)*exp(-(pow(x - x0, 2) + pow(y - y0, 2))/(2*pow(sigma,
        2)));
18             p[x][y] = r[x][y] + I*im[x][y];
19         }
20     }
21 }
```

As otimizações aplicáveis nessas funções são simples, consistem em reduzir o uso de funções pertencentes à biblioteca math.h quando deseja-se calcular o quadrado de determinada variável.

```
1  void potential(double **potential){
2      int i, j;
3      double n = 110;
4
5      for(i = 0; i < size; i++){
6          for(j = 0; j < size; j++)
7              potential[i][j] = 0.07*i*i + 0.01*j*j;
8
9      }
10 }
11
12 void wave_packet(double complex **p, double **im, double **r,
       double k0x, double k0y, double x0, double y0, double sigma){
13     int x, y, it = 0;
14
15     for(x = 0; x < size; x++){
16         for(y = 0; y < size; y++){
17             r[x][y] = cos(k0x*x + k0y*y)*exp(-(((x - x0)*(x - x0))
       + ((y - y0)*(y - y0)))/(2*(sigma*sigma)));
```

```
            im[x][y] = sin(k0x*x + k0y*y)*exp(-(((x - x0)*(x - x0))
    + ((y - y0)*(y - y0)))/(2*(sigma*sigma)));
            p[x][y] = r[x][y] + I*im[x][y];
        }

    }
}
```

8 Comparação do programa base com o otimizado

Considera-se novamente a matriz reduzida das análises anteriores.

8.1 Profile com otimizações nível código (sem flags)

```
real  29m15,086s
user  29m12,914s
sys 0m0,744s
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 96.86   1699.38  1699.38   600000     2.83     2.83
    finite_difference
  0.00   1699.40     0.02                              main
  0.00   1699.40     0.00        3     0.00     0.00  allocate
  0.00   1699.40     0.00        1     0.00     0.00
    allocate_complex
  0.00   1699.40     0.00        1     0.00     0.00  potential
  0.00   1699.40     0.00        1     0.00     0.00  wave_packet

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
     else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
     function is profiled, else blank.
```

```
33
34 name       the name of the function.  This is the minor sort
35           for this listing. The index shows the location of
36      the function in the gprof listing. If the index is
37      in parenthesis it shows where it would appear in
38      the gprof listing if it were to be printed.
39
40
41 Copyright (C) 2012-2020 Free Software Foundation, Inc.
42
43 Copying and distribution of this file, with or without modification
    ,
44 are permitted in any medium without royalty provided the copyright
45 notice and this notice are preserved.
46
47
48          Call graph (explanation follows)
49
50
51 granularity: each sample hit covers 2 byte(s) for 0.00% of 1699.40
    seconds
52
53 index % time    self  children    called     name
54                                                 <spontaneous>
55 [1]    100.0    0.02 1699.38                 main [1]
56              1699.38    0.00  600000/600000      finite_difference
    [2]
57                 0.00    0.00       3/3           allocate [3]
58                 0.00    0.00       1/1           allocate_complex
    [4]
59                 0.00    0.00       1/1           potential [5]
60                 0.00    0.00       1/1           wave_packet [6]
61 -----------------------------------------------
62              1699.38    0.00  600000/600000      main [1]
63 [2]    100.0 1699.38    0.00  600000         finite_difference [2]
64 -----------------------------------------------
65                 0.00    0.00       3/3           main [1]
66 [3]      0.0    0.00    0.00       3         allocate [3]
67 -----------------------------------------------
68                 0.00    0.00       1/1           main [1]
69 [4]      0.0    0.00    0.00       1         allocate_complex [4]
70 -----------------------------------------------
71                 0.00    0.00       1/1           main [1]
72 [5]      0.0    0.00    0.00       1         potential [5]
73 -----------------------------------------------
74                 0.00    0.00       1/1           main [1]
75 [6]      0.0    0.00    0.00       1         wave_packet [6]
76 -----------------------------------------------
77
78  This table describes the call tree of the program, and was sorted
    by
79  the total amount of time spent in each function and its children.
80
81  Each entry in this table consists of several lines.  The line with
     the
82  index number at the left hand margin lists the current function.
83  The lines above it list the functions that called this function,
```

```
and the lines below it list the functions this one called.
This line lists:
     index  A unique number given to each element of the table.
    Index numbers are sorted numerically.
    The index number is printed next to every function name so
    it is easier to look up where the function is in the table.

     % time This is the percentage of the `total' time that was
    spent
    in this function and its children.  Note that due to
    different viewpoints, functions excluded by options, etc,
    these numbers will NOT add up to 100%.

     self This is the total amount of time spent in this function.

     children This is the total amount of time propagated into this
    function by its children.

     called This is the number of times the function was called.
    If the function called itself recursively, the number
    only includes non-recursive calls, and is followed by
    a `+' and the number of recursive calls.

     name The name of the current function.  The index number is
    printed after it.  If the function is a member of a
    cycle, the cycle number is printed between the
    function's name and the index number.


For the function's parents, the fields have the following meanings
    :

     self This is the amount of time that was propagated directly
    from the function into this parent.

     children This is the amount of time that was propagated from
    the function's children into this parent.

     called This is the number of times this parent called the
    function `/' the total number of times the function
    was called.  Recursive calls to the function are not
    included in the number after the `/'.

     name This is the name of the parent.  The parent's index
    number is printed after it.  If the parent is a
    member of a cycle, the cycle number is printed between
    the name and the index number.

If the parents of the function cannot be determined, the word
`<spontaneous>' is printed in the `name' field, and all the other
fields are blank.

For the function's children, the fields have the following
    meanings:

     self This is the amount of time that was propagated directly
    from the child into the function.
```

```
138
139      children This is the amount of time that was propagated from
         the
140      child's children to the function.
141
142      called This is the number of times the function called
143      this child `/' the total number of times the child
144      was called.  Recursive calls by the child are not
145      listed in the number after the `/'.
146
147      name This is the name of the child.  The child's index
148      number is printed after it.  If the child is a
149      member of a cycle, the cycle number is printed
150      between the name and the index number.
151
152  If there are any cycles (circles) in the call graph, there is an
153  entry for the cycle-as-a-whole.  This entry shows who called the
154  cycle (as parents) and the members of the cycle (as children.)
155  The `+' recursive calls entry shows the number of function calls
         that
156  were internal to the cycle, and the calls entry for each member
         shows,
157  for that member, how many times it was called from other members
         of
158  the cycle.
159
160
161 Copyright (C) 2012-2020 Free Software Foundation, Inc.
162
163 Copying and distribution of this file, with or without modification
        ,
164 are permitted in any medium without royalty provided the copyright
165 notice and this notice are preserved.
166
167
168 Index by function name
169
170    [3] allocate               [2] finite_difference       [5]
         potential
171    [4] allocate_complex       [1] main                    [6]
         wave_packet
```

8.2 Análise de resultados

Como esperado, no profile do código otimizado, a função que mais consome processamento ainda é a função que aplica o método das diferenças finitas.

Também é possível observar que, tanto para o profile sem e com as flags, a otimização nas outras funções melhorou o desempenho do código. Embora o efeito dessa otimização tenha um impacto mais significativo sem as melhores flags, obtendo uma redução no tempo de execução de aproximadamente seis minutos, continua sendo mais vantajoso utilizar as melhores flags mesmo que o seu tempo tenha reduzido em apenas vinte segundos após as otimizações.

Portanto, a otimização do código por meio da aplicação das boas práticas de programação é indispensável para melhorar o desempenho do programa serial.

No entanto, essa técnica não é suficiente para reduzir o tempo de execução drasticamente como proposto, sendo necessário recorrer à outros artíficios (no caso, ao paralelismo) para melhorar o desempenho do programa.

9 Implementação do programa com CUDA

A paralelização do programa será realizada por intermédio da interface de progamação de aplicação CUDA para GPGPU criada pela Nvidia. As placas gráficas devem ser compátiveis com a API, isto é, geralmente essa devem possuir um chipset densenvolvido pela Nvidia. Abaixo está uma tabela com informações utéis sobre as plácas gráficas utilizadas no projeto.

	CUDA cores	Arquitetura	Memória	DPFLOPS
Tesla K40t	2880	Volta	12GB	1.43
Tesla V100	5120	Volta	16-32GB	7.8

Tabela 1: Tabela com as informações de cada GPU a ser utilizada na implementação CUDA desse programa. Os valores dara Double Precision FLOPS (DPFLOPS) são dados em TeraFLOPS.

Como é possível observar, a segunda placa gráfica possui uma quantidade maior de DPFLOPS, o que pode indicar um melhor desempenho em relação à primeira.

9.1 Estratégia de paralelização

A ideia central da paralelização desse problema concentra-se na conversão de uma matriz NxN por um vetor de tamanho equivalente. Deste modo, a estratégia adotada utiliza de N blocos com N threads cada, onde associa-se os blocos com as linhas da matriz e as threads com as colunas. Todos os blocos pertencem ao mesmo grid para que seja possível realizar a comunicação entre os dados de dependência temporal e a dimensão do bloco servirá para orientar a posição do bloco na grade.

```
1       matrix[i][j] -> matrix[i * size + j]
2 matrix[i*size + j] -> matrix[blockIdx.x * blockDim.x + threadIdx.x]
```

9.2 Profile

O profile para o código paralelizado com CUDA leva em consideração tanto a análise a nível de API quanto para a nível de GPU, uma vez que o comando utilizado foi:

```
1             nvprof -s -o schrodinger.nvprof ./schrodinger.x
```

Logo, segue as informações do profile.

```
[bianca.gomes@sdumont14 cuda]$ cat profile.txt
==28954== NVPROF is profiling process 28954, command: ./schrodinger.x
==28954== Error: File already exists: /scratch/uff21hpc/bianca.gomes/cuda/schrodinger.nvprof
==28954== Use "-f" to overwrite existing file
==28954== Profiling application: ./schrodinger.x
==28954== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   97.86%  36.1716s     19250  1.8790ms  1.6329ms  2.3578ms  [CUDA memcpy HtoD]
                    2.14%  792.57ms      2750  288.21us  285.51us  292.55us  finite_difference_kernel
      API calls:   99.27%  42.5558s     38500  1.1053ms  2.0310us  2.8927ms  cudaMemcpy
                    0.39%  165.20ms         7  23.600ms  133.63us  164.36ms  cudaMalloc
                    0.31%  133.46ms      2750  48.532us  42.264us  301.91us  cudaLaunchKernel
                    0.02%  7.9675ms     19250     413ns     255ns  617.62us  cudaGetErrorString
                    0.01%  4.3994ms         7  628.49us  326.04us  688.15us  cudaFree
                    0.00%  912.90us         2  456.45us  446.80us  466.11us  cuDeviceTotalMem
                    0.00%  542.22us         2  271.11us  250.15us  292.07us  cudaGetDeviceProperties
                    0.00%  509.24us       194  2.6240us     147ns  129.18us  cuDeviceGetAttribute
                    0.00%  49.971us         2  24.985us  23.498us  26.473us  cuDeviceGetName
                    0.00%  14.285us         2  7.1420us  2.6400us  11.645us  cuDeviceGetPCIBusId
                    0.00%  6.3310us         1  6.3310us  6.3310us  6.3310us  cudaGetDeviceCount
                    0.00%  3.8690us         4     967ns     200ns  2.8780us  cuDeviceGet
                    0.00%  1.9860us         3     662ns     160ns  1.1910us  cuDeviceGetCount
                    0.00%     625ns         2     312ns     212ns     413ns  cuDeviceGetUuid
```

Figura 10: Profile para o código em CUDA. É fácil observar que a maior parte das atividades estão nas funções de cópia de *host* para *device* e vice-versa nomeadas cudaMemcpy.

10 Benchmark (paralelo)

Preliminarmente, é essencial a informação de que para as análises a seguir o tempo final e a matriz do programa foram reduzidos. Essa redução foi necessária por causa do tempo das filas de submissão dos clusters utilizados.

O tempo final será sempre

```
tempo final : 400
```

e o tamanho da matriz irá variar entre

```
cudacores="2880 5760 8640"
```

na finalidade de se ajustar aos cudacores em relação a GPU Tesla K40t e sem ultrapassar o tempo das filas de submissão.

10.1 Tesla K40t (LNCC)

Decorrente da sobrecarga de comunicação entre *host* e *device* e também da quantidade de DPFLOPS dessa placa gráfica, o tempo do programa paralelizado ultrapassa o tempo serial para todos os casos.

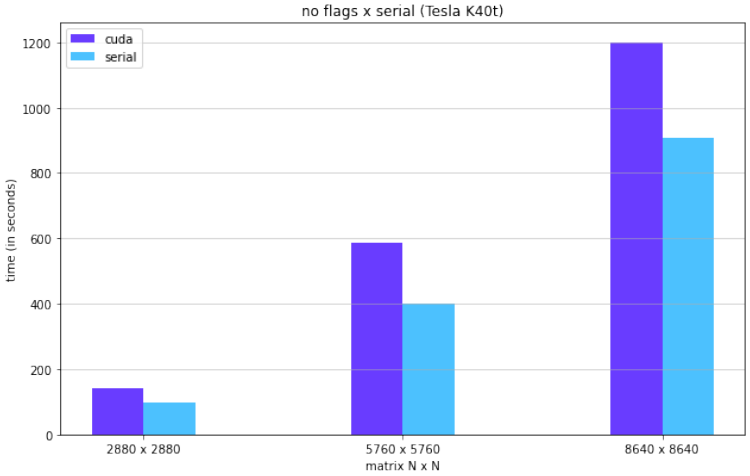

Figura 11: Benchmark para a comparação entre os tempos do programa serial e os tempos para o paralelizado com CUDA sem utilizar flags.

A eficiência é pior para o segundo caso onde há uma matriz 5760 x 5760, pode-se considerar que a relação entre o tamanho da malha e a quantidade de cudacores da GPU está prejudicando com processos de *pipeline*. De qualquer forma, para todos os casos, o programa é simplesmente ineficiente.

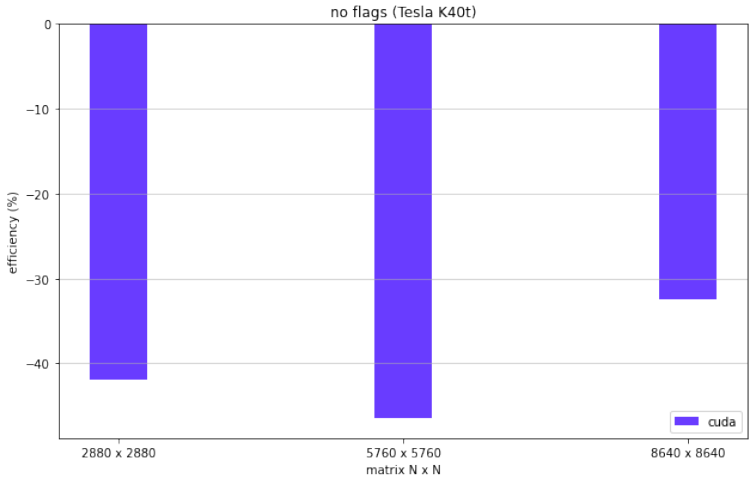

Figura 12: Benchmark com a eficiência do programa na GPU Tesla K40t em comparação com o tempo do programa serial. Ambos estão sem nenhuma flag.

O benchmark do speed up é completamente coerente para um programa ineficiente. Não há ganho em tempo, pelo contrário, é preferível executar o serial do que o paralelo. Enquanto que o parâmetro para o gráfico plano é a reta x = y, o parâmetro para o histograma é o valor um, portanto é razoável que todos os valores marcados sejam menores que esse número.

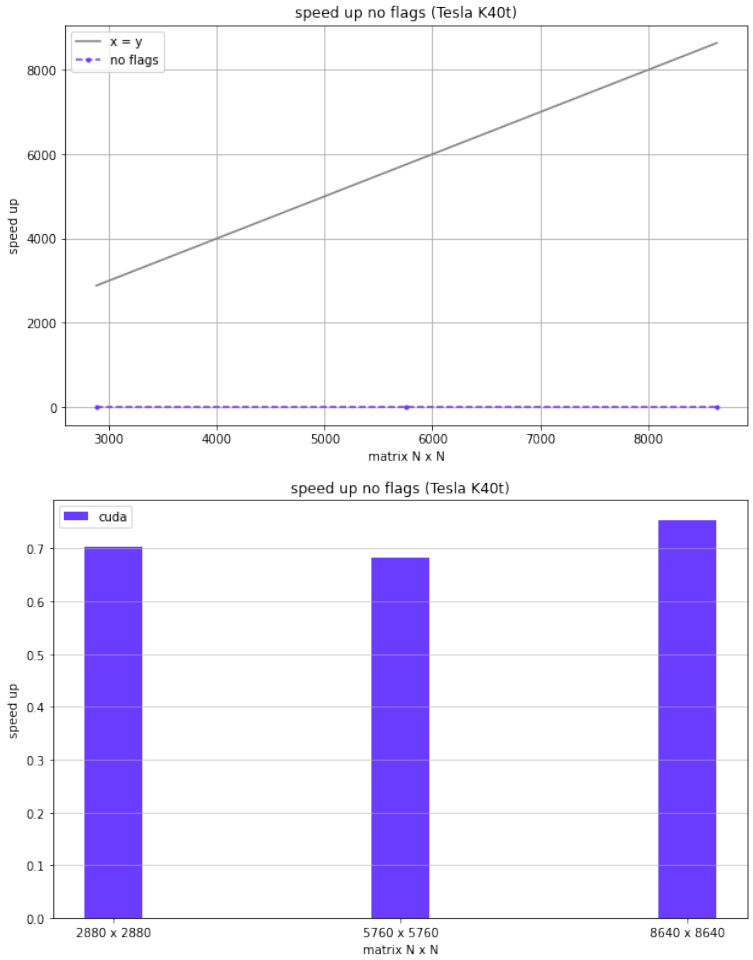

Figura 13: Benchmark para o speed up do programa paralelizado. Sem nenhuma flag.

10.1.1 Análise de flags

As flags utilizadas para aumentar a eficiência da implementação em CUDA estão dispostas abaixo. Todas estão acompanhadas de uma flag de otimização $-OX$ para X de 1 até 3.

```
declare -a flagscuda=(
        "1" " "
        "2" "-Xptxas"
        "3" "-use_fast_math"
)
```

A segunda flag refere-se a aplicação das otimizações no *device*.

No primeiro caso, que é para a primeira configuração de matriz, as flags do *device* são as piores para todas as otimizações, infere-se que é por conta das operações que ocorrem no *kernel*. Não há grandes diferenças entre os tempos do primeiro e terceiro conjunto de flags, portanto considera-se a melhor flags sendo

```
-O3 -use_fast_math
```

nessa situação.

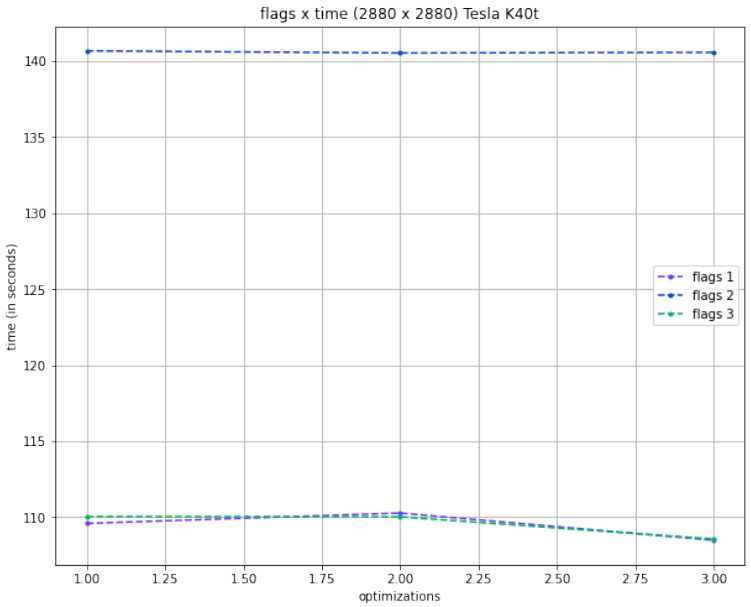

Figura 14: Benchmark do tempo para o conjunto de flags na matriz 2880 x 2880.

As melhores flags para a matriz 5760 x 5760 são as mesmas da análise anterior. No entanto, nota-se aqui um comportamento peculiar na otimização -O2, uma espécie de inversão de desempenho entre as flags de otimização do *host* e *device*. A melhor suposição é de que tal comportamento é decorrente do *pipeline*.

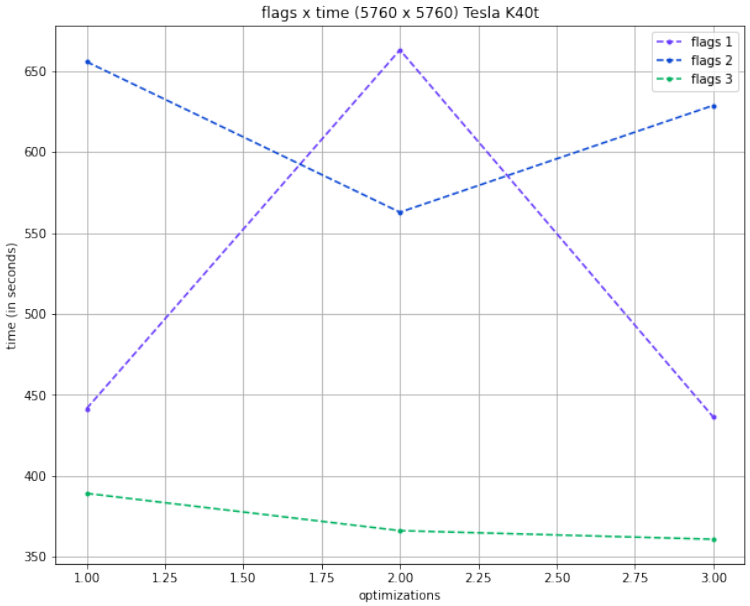

Figura 15: Benchmark do tempo para o conjunto de flags na matriz 5760 x 5760.

Novamente, as melhores flags são as mesmas das duas análises anteriores. O comportamento de inversão entre *host* e *device* aparece novamente.

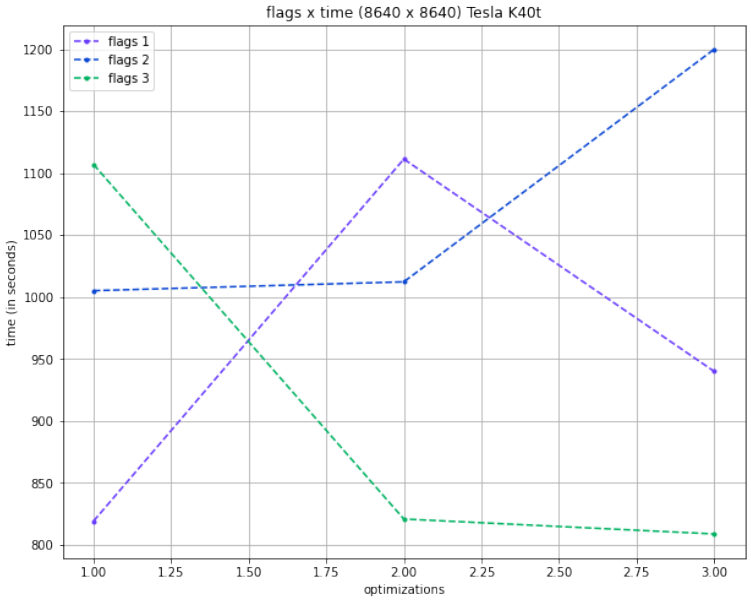

Figura 16: Benchmark do tempo para o conjunto de flags na matriz 8640 x 8640.

No geral, não há muito o que comentar sobre as eficiências separadamente pois essas se comportam de acordo com a análise temporal. O relevante aqui é enfatizar que, para a placa Tesla K40t, o programa com a matriz 5760 x 5760 unido das melhores flags atingiu a maior eficiência que está em torno de 40%.

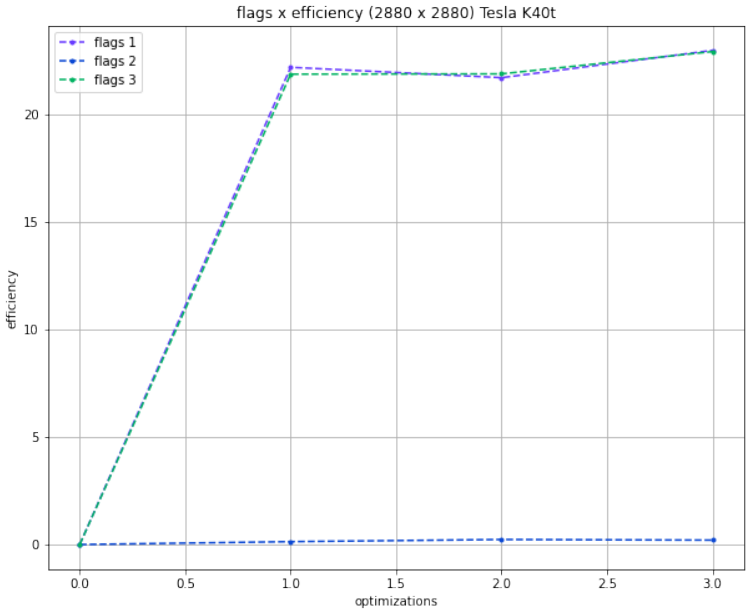

Figura 17: Benchmark da eficiência para o conjunto de flags na matriz 2880 x 2880.

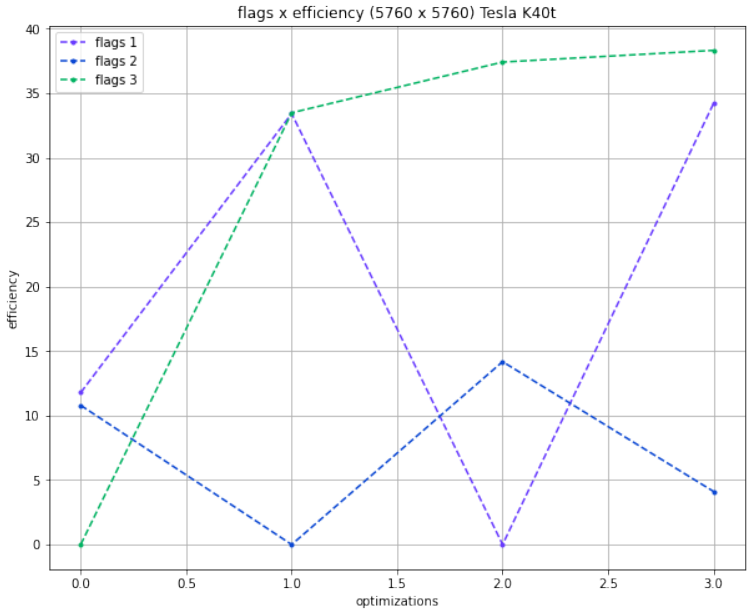

Figura 18: Benchmark da eficiência para o conjunto de flags na matriz 5760 x 5760.

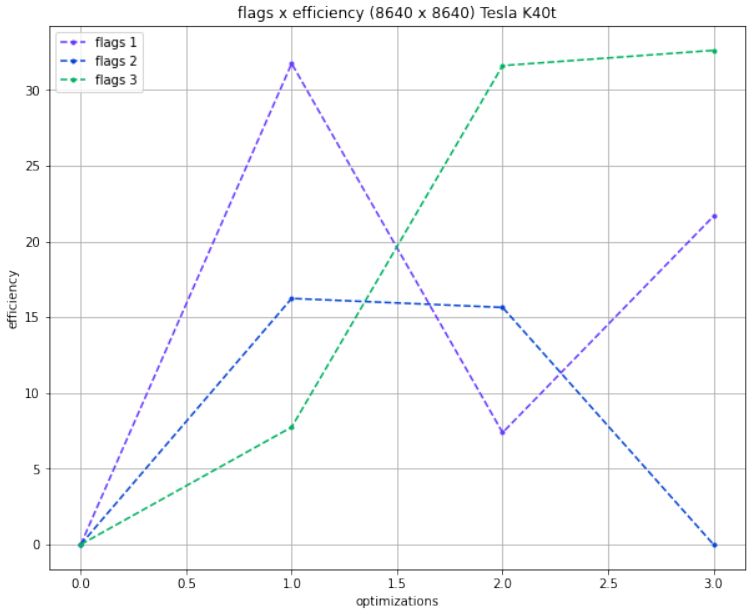

Figura 19: Benchmark da eficiência para o conjunto de flags na matrix 8640 x 8640.

10.2 Tesla V100 (LNCC)

Nessa placa gráfica, nota-se uma melhoria do tempo de execução do programa paralelo em relação ao serial. Esse melhor desempenho deve-se à quantidade de DPFLOPS presentes nessa placa.

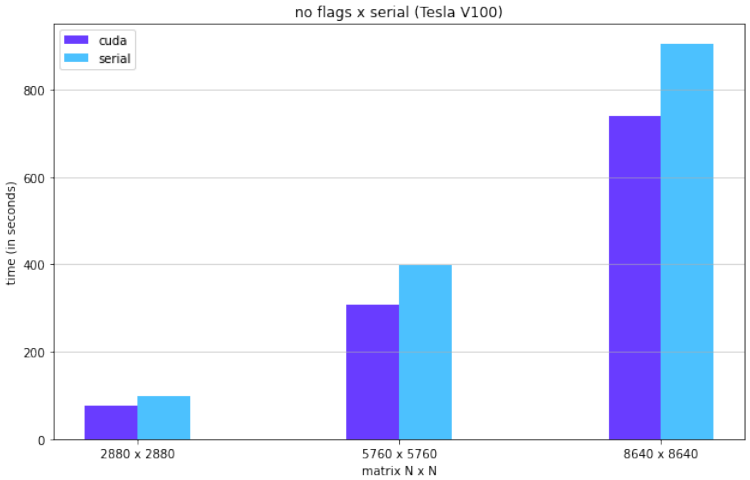

Figura 20: Benchmark dos tempos do programa paralelizado em comparação com os tempos do programa serial.

Ainda assim, a eficiência é considerada pouca, uma vez que não chega nem em 50%. As melhores eficiências são correspondentes às duas primeiras configurações de matriz, considera-se que isso acontece por causa da quantidade de CUDA cores na placa ser maior do que a primeira matriz e apenas um pouco menor do que a segunda, de tal modo que o *pipeline* não é tão agressivo para o desempenho do programa. No entanto, é nítido que o mesmo não aconteceu para a terceira configuração de matriz e para esse o programa é nitidamente menos eficiente.

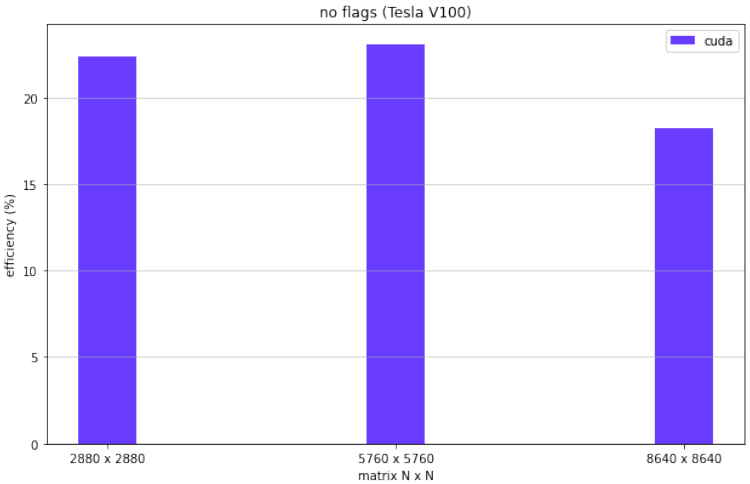

Figura 21: Benchmark da eficiência do programa paralelizado em CUDA.

Há ganho em eficiência no programa, mas é pequeno. Nota-se a distância entre o speed up e a reta de referência, como também nota-se que o aumento é decimal com relação ao número de referência do histograma.

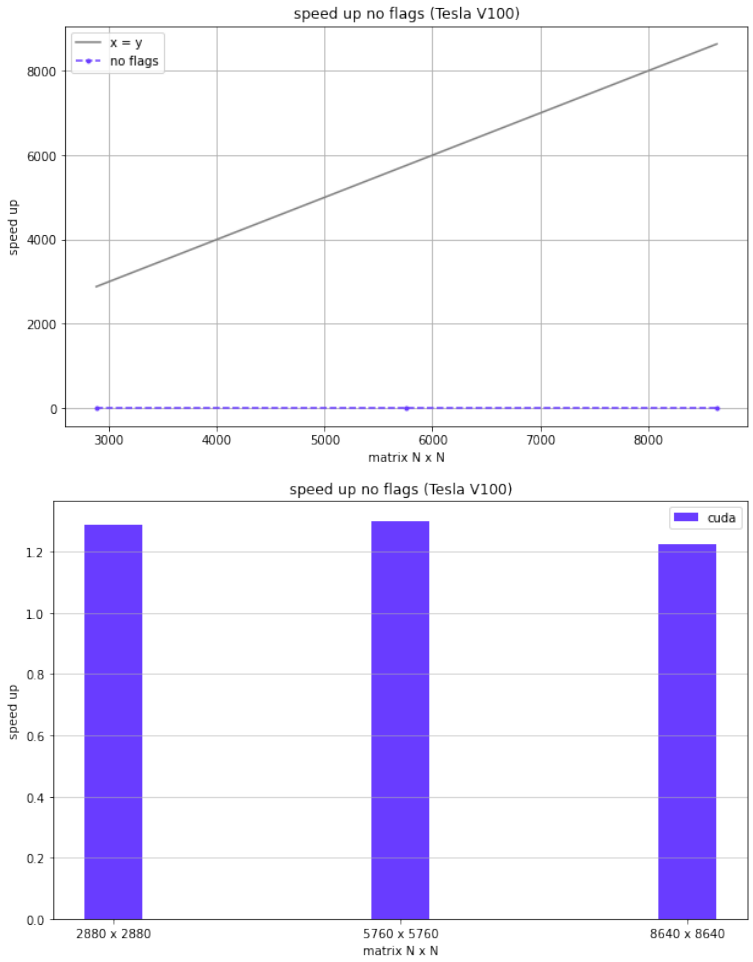

Figura 22: Benchmark do speed up do programa parelelizado executado com a placa Tesla V100.

10.2.1 Análise de flags

As mesmas flags da análise para a placa de vídeo anterior estão sendo utilizadas.

Semelhante à situação na outra placa gráfica, a flag do *device* é a pior para todas as otimizações. No entanto, aqui tem-se que a melhor flag é simplesmente a otimização -O3.

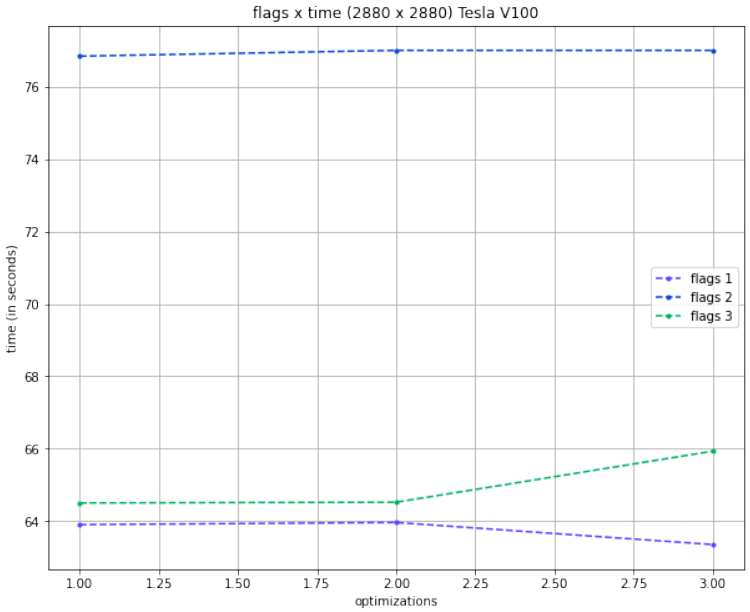

Figura 23: Benchmark do tempo para o conjunto de flags na matriz 2880 x 2880.

As melhores flags ficam entre as que carregam a otimização -O3 do primeiro e terciero conjunto de flags. Consideraremos que a melhor é -O3 por causa da situação na matriz 2880 x 2880. Não há nenhum comportamento atípico nas flags do *device*, o que pode ser atríbuido como consequência da quantidade de cudacores e DPFLOPS da placa gráfica.

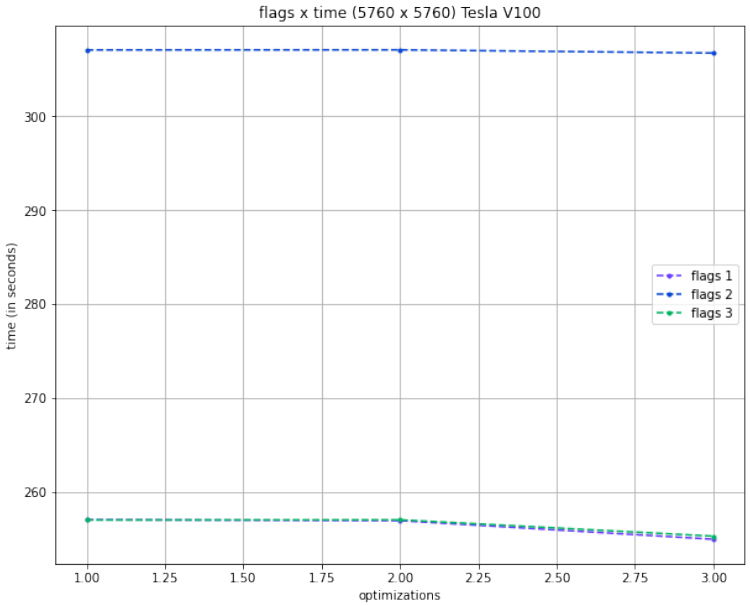

Figura 24: Benchmark do tempo para o conjunto de flags na matriz 5760 x 5760.

No entanto, para a matriz de tamanho 8640 x 8640, a melhor flag é

```
-O3 -use_fast_math
```

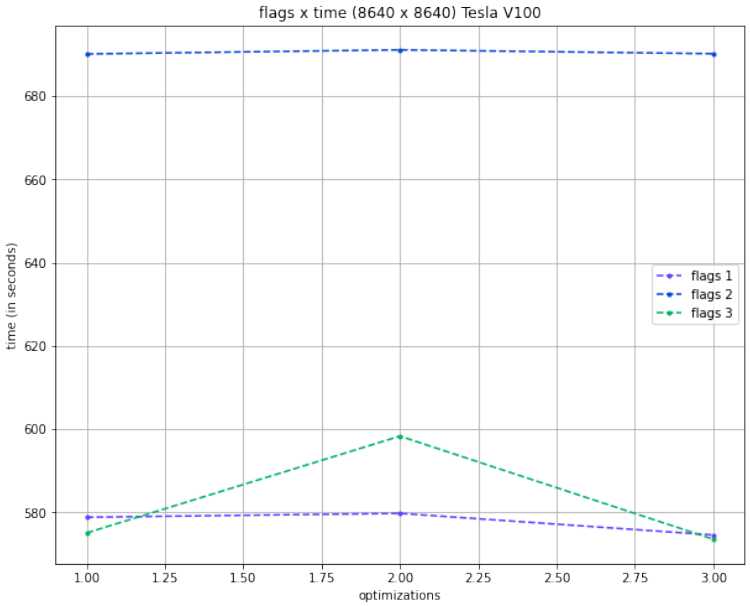

Figura 25: Benchmark do tempo para o conjunto de flags na matriz 8640 x 8640.

Novamente, como as eficiências seguem o comportamento do desempenho temporal, o importante a ser destacado é que a configuração de matriz mais eficiente é a 8640 x 8640.

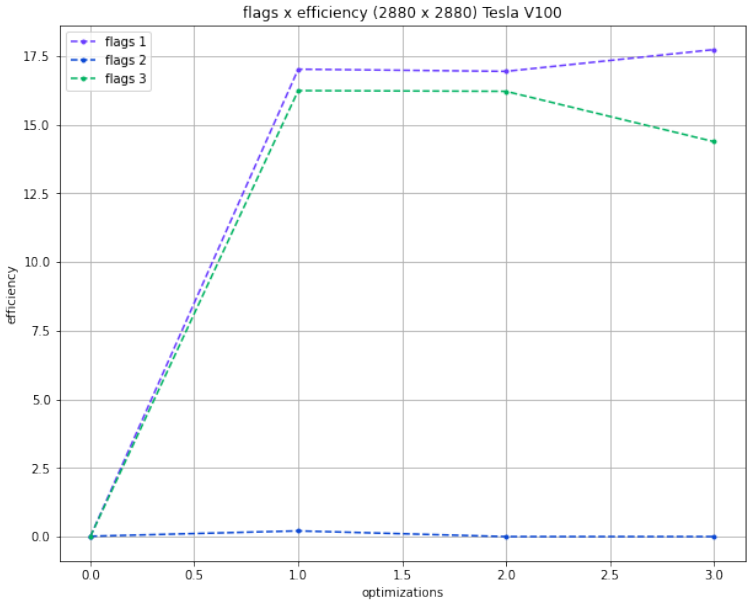

Figura 26: Benchmark da eficiência para o conjunto de flags na matriz 2880 x 2880.

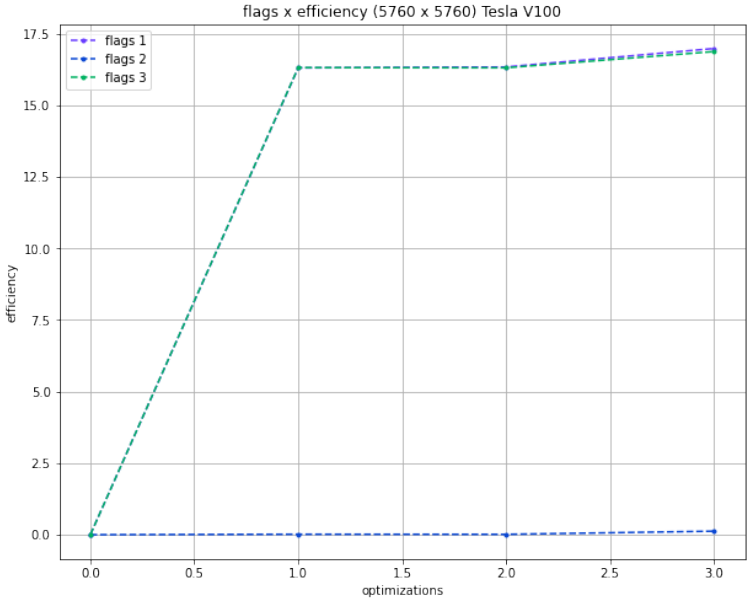

Figura 27: Benchmark da eficiência para o conjunto de flags na matriz 5760 x 5760.

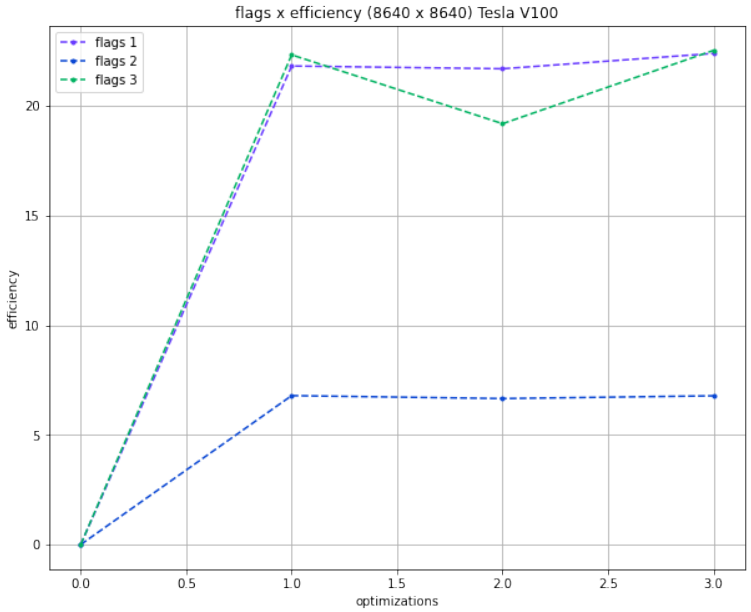

Figura 28: Benchmark da eficiência para o conjunto de flags na matriz 8640 x 8640.

10.3 Benchmark com as melhores flags (Tesla V100)

Dessa vez, com as melhores flags, os tempos obtidos para o código em CUDA são melhores do que os tempos do código em serial com uma diferença satisfatória no tempo de execução.

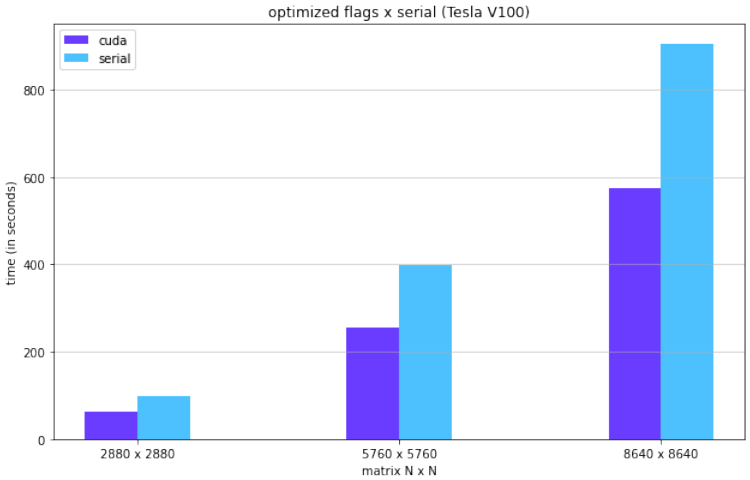

Figura 29: Benchmark da comparação entre os tempos otimizados com flags do código paralelo e os tempos do código serial.

A eficiência é nivelada apresentando valores semelhantes para todos os conjuntos de matrizes analisados.

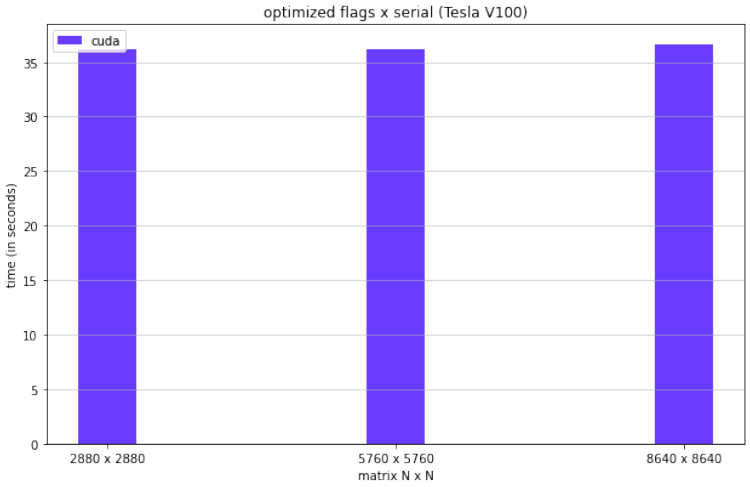

Figura 30: Benchmark da comparação entre as eficiências otimizadas com flags do código paralelo e os tempos do código serial.

11 Comparação com a implementação em OpenMP

O benchmark abaixo demonstra a evolução da eficiência do programa que anteriormente foi paralelizado com OpenMP. A execução desse programa foi realizado em um computador caseiro que tem as mesmas configurações que o computador do Laboratório 107C da UFF.

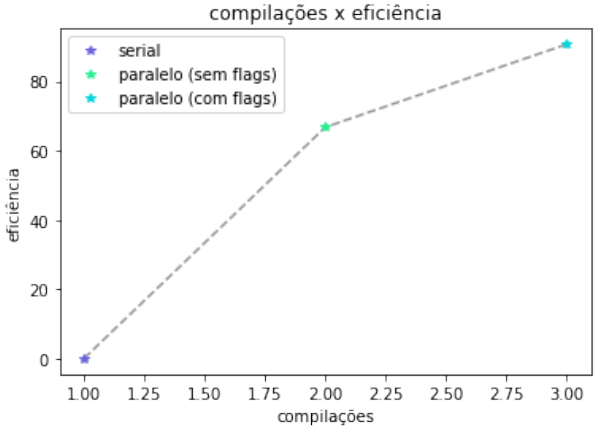

Figura 31: Gráfico para o desenvolvimento da eficiência por compilação do programa paralelizado com OpenMP.

O benchmark para o programa paralelizado em CUDA leva em consideração a matriz da melhor eficiência obtida com o uso de flags.

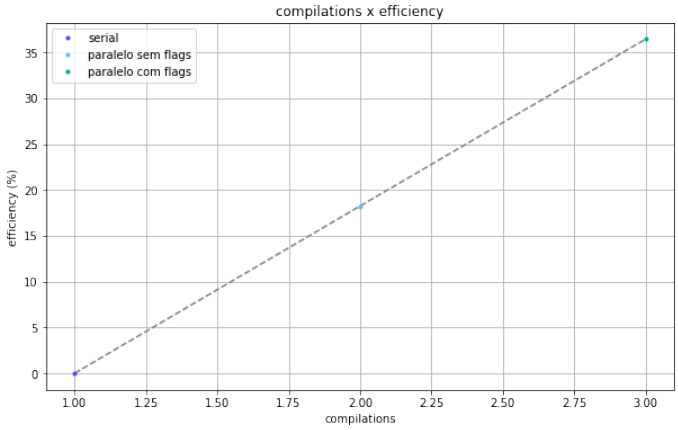

Figura 32: Gráfico para o desenvolvimento da eficiência por compilação do programa paralelizado com CUDA. A matriz utilizada foi a de tamanho 8640 x 8640 pois essa obteve a melhor eficiência com flags. Os dados referem-se a execução do programa na placa Tesla V100.

A conclusão de que o programa em OpenMP é superior em questão de desempenho temporal é imediato, pois enquanto esse obteve uma eficiência em torno de $80 - 90\%$ em um computador caseiro, o programa em CUDA obteve em torno de 35% de eficiência em uma placa gráfica voltada para computação de alto desempenho.

11.1 Considerações sobre o método das diferenças finitas

É evidente pelos resultados obtidos nos benchmarks que essa estratégia de paralelização adotada é ineficiente ou pouco eficiente.

O método das diferenças finitas como foi aplicado nessa solução para o problema da equação de Schrödinger, obrigatoriamente exige a separação entre as partes reais e imaginárias da equação. Consequentemente, isso sobrecarrega a quantidade de informação que deve ser comunicada entre o *host* e o *device*, que é possível confirmar a ocorrência ao observar o profile do código.

Uma alternativa em potencial para resolver esse problema de eficiência é substituir o método número vigente por algum método que utilize da *Fast Fourier Transform*. Obviamente, não descarta-se a possibilidade de outra estratégia de paralelização com o método atual que seja mais eficiente. No entanto, a eficácia da FFT com aplicações em CUDA é reconhecida, existindo até mesmo uma *library* da Nvidia para isso [3].

12 Validação de Resultados

Dos postulados da mecânica quântica tem-se a declaração essencial de que a equação de Schrödinger é responsável pela evolução temporal do sistema. Com isso, é equivalente dizer, tendo em mente a interpretação probabilística, que a normalização da função de onda se conserva com o tempo. Portanto, para validar os resultados da implementação computacional para a solução dessa equação, considera-se a densidade de probabilidade descrita pela equação abaixo.

$$P(x, y, t)dxdy = |\psi(x, y, t)|^2 dxdy \tag{9}$$

Com os resultados de compilações do programa para uma matriz reduzida e tempos finais diferentes, obteve-se as seguintes imagens:

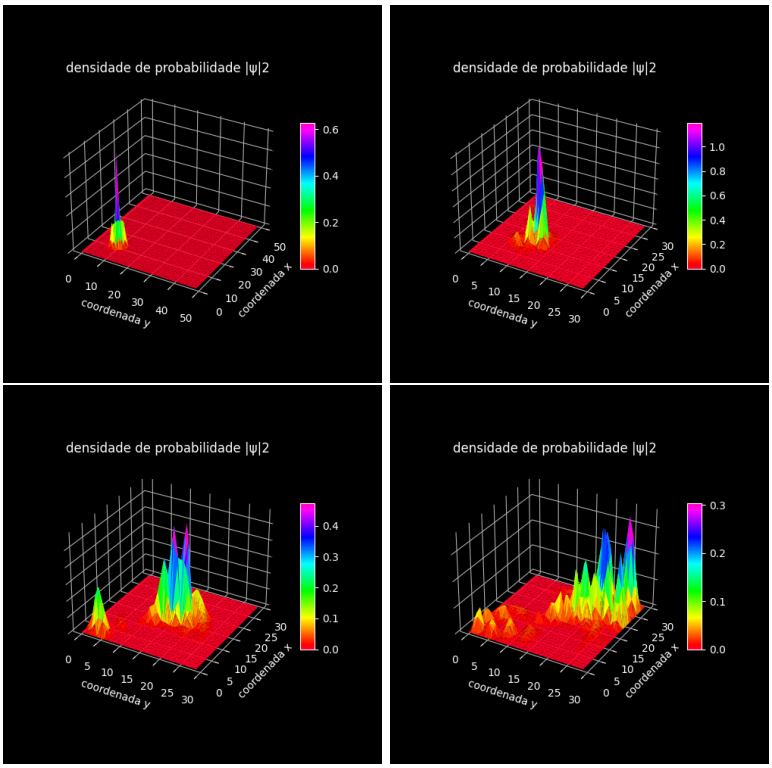

Figura 33: Representação gráfica obtida com a compilação do programa serial para tempos finais diferentes, respectivamente, tf=0, tf=100000, tf=500000 e tf=1000000. Nesse caso, a matriz está reduzida.

Com essas imagens nota-se que a função de onda transita pelo potencial conforme ocorre a evolução temporal do sistema. Outra importantissíma

característica é a preservação da distruibuição gaussiana, tanto para a parte transmitida quanto para a parte refletida, conservando a interpretação probabilística.

Para confirmar a preservação dos dados após a paralelização do programa e também para que não seja necessário uma avaliação trabalhosa de dado por dado, ainda considerando a interpretação probabilística do problema tem-se como parâmetro a média e o desvio padrão dos dados.

$$\mu = \frac{\sum_i^n x_i}{n} \tag{10}$$

$$\sigma = \sqrt{\frac{\sum_i^n (x_i - \overline{x})^2}{n}} \tag{11}$$

Implementação	Média μ	Desvio Padrão σ
Serial	$8.00 \cdot 10^{-8}$	$1.78 \cdot 10^{-5}$
CUDA	$1.56 \cdot 10^{-7}$	$2.80 \cdot 10^{-5}$

Tabela 2: Tabela com os valores de média e desvio padrão para cada implementação do programa.

O critério de qualidade desses dados é definido pela discrepância dos dados do código em paralelo com os dados do código em serial.

	Discrepância da média	Discrepância do desvio padrão
CUDA	$7.65 \cdot 10^{-8}$	$1.02 \cdot 10^{-5}$

Tabela 3: Tabela com as discrepâncias entre os dados do código paralelizado e sua versão serial.

Como as discrepâncias entre os dois conjuntos de dados são extremamente pequenas, pode-se considerar que a implementação da paralelização em CUDA não alterou a qualidade dos dados e que a solução do problema foi efetivada com sucesso.

13 Conclusões

Com o auxílio das melhores flags obtidas com os benchmarks para o programa serial realizou-se o profile, onde ficou claro que a função responsável pelo método das diferenças finitas consumia mais processamento que todas as outras e que a paralelização deveria atuar diretamente nessa função e no loop onde a mesma é chamada.

O método das diferenças finitas não aparenta ser o melhor método para receber uma implementação CUDA do problema abordado, dado que esse exige a separação entre as partes real e imaginária da equação de onda quântica e portanto, sobrecarregando a comunicação entre *host* e *device*. Ainda assim, com a placa Tesla V100 obteve-se eficiência por mais que seja pouca. No entanto, com a placa Tesla K40t, o programa foi simplesmente ineficiente inferindo que será sempre necessário uma placa com foco em computação de alto desempenho para que o programa seja minimamente eficiente.

Finalmente, diante da validação dos resultados, percebendo que as discrepâncias entre as médias e desvios padrões do programa serial e do programa paralelizado são extremamente pequenas, pode-se concluir que a paralelização foi concluída corretamente, embora os ganhos em desempenho temporal não sejam satisfatórios.

14 Apêndice

14.1 Informações sobre as CPUs

14.1.1 Laboratório 107C (UFF)

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 42
Model name:            Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
Stepping:              7
CPU MHz:               1599.975
CPU max MHz:           3800,0000
CPU min MHz:           1600,0000
BogoMIPS:              6784.74
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
sse2 ssht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon
pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf
eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2
ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt
tsc_deadline_timer aes xsave avx lahf_lm epb ssbd ibrs ibpb
stibp tpr_shadow vnmi flexpriority ept vpid xsaveopt dtherm
ida arat pln pts md_clear spec_ctrl intel_stibp flush_l1d
```

14.1.2 B107 (LNCC)

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
```

```
Stepping:              4
CPU MHz:               2899.072
CPU max MHz:           3200,0000
CPU min MHz:           1200,0000
BogoMIPS:              4800.35
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-11
NUMA node1 CPU(s):     12-23
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx
smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2
x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand
lahf_lm epb ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority
ept vpid fsgsbase smep erms xsaveopt dtherm ida arat pln pts
md_clear spec_ctrl intel_stibp flush_l1d
```

14.1.3 SequanaX (LNCC)

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                96
On-line CPU(s) list:   0-47
Off-line CPU(s) list:  48-95
Thread(s) per core:    1
Core(s) per socket:    24
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz
Stepping:              7
CPU MHz:               2101.000
CPU max MHz:           2101,0000
CPU min MHz:           1000,0000
BogoMIPS:              4200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              36608K
NUMA node0 CPU(s):     0-23
NUMA node1 CPU(s):     24-47
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx
```

```
smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1
sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
f16c rdrand lahf_lm abm 3dnowprefetch epb cat_l3 cdp_l3
invpcid_single intel_ppin intel_pt ssbd mba ibrs ibpb stibp
ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid fsgsbase
tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx
rdt_a avx512f avx512dq rdseed adx clflushopt clwb avx512cd
avx512bw avx512vlxsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc
cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts hwp
hwp_act_window hwp_epp hwp_pkg_req pku ospke avx512_vnni
md_clear spec_ctrl intel_stibp flush_l1d arch_capabilities
```

14.2 Informações sobre as GPUs

14.2.1 Tesla K40t

```
device 0 information
       name: Tesla K40t
       compute capability: 3.5
       clock rate: 745000
       device copy overlap: enabled

       kernel execution timeout:
       disabled

device memory information
       total global memory: 11997020160
       total constant memory: 65536
       max memory pitch: 2147483647
       texture alignment: 512

device multiprocessor information
       multiprocessor count: 15
       shared memory per multiprocessor: 49152
       register per multiprocessor: 65536
       threads in warp: 32
       max thread dimensions: (1024, 1024, 64)
       max grid dimensions: (2147483647, 65535, 65535)


device 1 information
       name: Tesla K40t
       compute capability: 3.5
       clock rate: 745000
       device copy overlap: enabled

       kernel execution timeout:
       disabled

device memory information
       total global memory: 11997020160
       total constant memory: 65536
       max memory pitch: 2147483647
       texture alignment: 512

device multiprocessor information
```

```
        multiprocessor count: 15
        shared memory per multiprocessor: 49152
        register per multiprocessor: 65536
        threads in warp: 32
        max thread dimensions: (1024, 1024, 64)
        max grid dimensions: (2147483647, 65535, 65535)
```

14.2.2 Tesla V100

```
device 0 information
        name: Tesla V100-SXM2-32GB
        compute capability: 7.0
        clock rate: 1530000
        device copy overlap: enabled

        kernel execution timeout:
        disabled

device memory information
        total global memory: 34089730048
        total constant memory: 65536
        max memory pitch: 2147483647
        texture alignment: 512

device multiprocessor information
        multiprocessor count: 80
        shared memory per multiprocessor: 49152
        register per multiprocessor: 65536
        threads in warp: 32
        max thread dimensions: (1024, 1024, 64)
        max grid dimensions: (2147483647, 65535, 65535)


device 1 information
        name: Tesla V100-SXM2-32GB
        compute capability: 7.0
        clock rate: 1530000
        device copy overlap: enabled

        kernel execution timeout:
        disabled

device memory information
        total global memory: 34089730048
        total constant memory: 65536
        max memory pitch: 2147483647
        texture alignment: 512

device multiprocessor information
        multiprocessor count: 80
        shared memory per multiprocessor: 49152
        register per multiprocessor: 65536
        threads in warp: 32
        max thread dimensions: (1024, 1024, 64)
        max grid dimensions: (2147483647, 65535, 65535)

```

```
48
49 device 2 information
50        name: Tesla V100-SXM2-32GB
51        compute capability: 7.0
52        clock rate: 1530000
53        device copy overlap: enabled
54
55        kernel execution timeout:
56        disabled
57
58 device memory information
59        total global memory: 34089730048
60        total constant memory: 65536
61        max memory pitch: 2147483647
62        texture alignment: 512
63
64 device multiprocessor information
65        multiprocessor count: 80
66        shared memory per multiprocessor: 49152
67        register per multiprocessor: 65536
68        threads in warp: 32
69        max thread dimensions: (1024, 1024, 64)
70        max grid dimensions: (2147483647, 65535, 65535)
71
72
73 device 3 information
74        name: Tesla V100-SXM2-32GB
75        compute capability: 7.0
76        clock rate: 1530000
77        device copy overlap: enabled
78
79        kernel execution timeout:
80        disabled
81
82 device memory information
83        total global memory: 34089730048
84        total constant memory: 65536
85        max memory pitch: 2147483647
86        texture alignment: 512
87
88 device multiprocessor information
89        multiprocessor count: 80
90        shared memory per multiprocessor: 49152
91        register per multiprocessor: 65536
92        threads in warp: 32
93        max thread dimensions: (1024, 1024, 64)
94        max grid dimensions: (2147483647, 65535, 65535)
```

Referências

[1] Jos Thijssen. *Computational physics*. Cambridge university press, 2007. Citado na página 6.

[2] H Rubin and Author Landau. *Computational Physics: Problem Solving with Computers*. J. Wiley & sons, 1997. Citado na página 6.

[3] Fast fourier transforms for nvidia gpus. https://developer.nvidia.com/cufft. Citado na página 46.